NOKIA bell com

Tech Talk: Introduction to Rust language

A systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety

- Pierre-Henri Symoneaux
- 07-09-2017

The language in a few words

- A low level language compiled to CPU instructions (no bytecode)
- LLVM as the compiler backend. Supports many targets
- A powerful and very strict compiler performing many checks for safety
- Binaries are statically linked by default
- No Garbage Collector

Major features are :

- Zero-cost abstractions
- Move semantic
- Guaranteed **memory safety**
- Threads without data races
- Trait-based generics
- Pattern matching
- Type inference
- Minimal runtime
- Efficient C bindings

A bit of history

- Started as a personal project by Graydon Hoare in 2006 while working at Mozilla
- Mozilla started to contribute in 2009
- First official announcement made in 2010
- First alpha release in 2012
- Samsung joined the community in 2013 (working on the Servo engine)
- Stable 1.0 release in May 2015
- Current release from September 2017 is 1.20.0

A growing popularity

- Ranked 40th on May 2017 Tiobe Index
- Rustlang's GitHub page has around 1100 watchers, 21600 stars and 4000 forks
- Around 10000 libraries indexed on crates.io for a total of 150 000 000 downloads
- More and more influent adopters (see https://www.rust-lang.org/friends.html):
  - Canonical, Chef, OVH, Dropbox, NPM, Samsung, Mozilla, Gnome, and more …
- According to Stackoverflow's 2017 developer survey:
  - Rust is the **Most Loved Language** for the 2nd year
  - Rust is the 2nd top paying technology worldwide

Release channels

- Rust has 3 release channels
  - Stable, Beta and Nightly
  - Stable release happen every 6 weeks (usually on Thursday)

- The nightly release has unstable/experimental features not available in beta and stable releases
  - In the language syntax itself
  - In the standard library

- This presentation is not about unstable and experimental features and is only based on **stable release 1.20.0**

The ecosystem and the community

- **Rustup** : the rust installer, can switch between stable, beta and nightly releases
- **Cargo** and **crates.io** : package/project manager, is to Rust what NPM is to node.js
- **Rustfmt** and **clippy** : tools for formatting and linting the code
- **Racer** : a tool used by editors for autocompletion
- A set of very good documentations (standard library, books, guides)
  - See https://doc.rust-lang.org/
    - The Rust Language book
    - The Unstable book
    - The Rustonomicon
- **This week in Rust** : a weekly blog about news from the community
- Code editors
  - Atom, VSCode, Sublime Text, Eclipse, and more … are well supported

NOKIA bcom

The Rust Language

Basic features and syntax

Basic features and syntax
Hello World !!!

- In the file **hello_world.rs**, write

```
/// This is the program entry point
fn main() {
    println!("Hello World !!!");
}
```

- Compile by running **`rustc hello_world.rs`**
- Run **hello_world** or **hello_world.exe**

Basic features and syntax

Just to name a few of them

- A syntax similar to C with block delimited by curly brackets
- Conditional control with keywords if, else, while, for, ...
- The match keyword is similar to switch in C, with some enhancements
- Primitive types like i32, i64, u32, u64, u8, f32, f64
- Native Unicode strings
- Immutability by default
- Enums
- There's no class, only structures
- Methods can be defined for structures
- Support pointers
- Namespaces
- Unit tests natively supported

Basic features and syntax
A short example

```
/// A simple counter
pub struct Counter {
	cnt: u64
}

impl Counter {
	/// Creates a new counter initialized to 0
	pub fn new() -> Counter {
		return Counter {
			cnt: 0
		};
	}

	/// Get the current counter value
	pub fn get_value(&self) -> u64 {
		self.cnt // last expression is
	returned if not ending by semicolon
	}

	/// Increments the counter value
	pub fn incr(&mut self) {
		self.cnt += 1;
	}
}
```

```
pub fn main() {
	let mut counter = Counter::new();
	println!("value = {}", counter.get_value()); // Prints
	"value = 0"
	counter.incr();
	println!("value = {}", counter.get_value()); // Prints
	"value = 1"
}

#[cfg(test)]
mod tests {
	use super::Counter;

	#[test]
	fn it_works() {
		let mut counter = Counter::new();
		assert_eq!(counter.get_value(), 0);
		counter.incr();
		assert_eq!(counter.get_value(), 1);
	}
}
```


Basic features and syntax
Embedded documentation

- Rust support for embedded doctrings
- A doctring starts with ***///***
- Doctrings support Markdown syntax
- Embedded Rust code examples can be tested by compiling and running them
- Generated HTML doc is well designed and easy to browse

Struct bcom::Counter

```
pub struct Counter { /* fields omitted */ }
```

[−] A simple counter

Methods

impl Counter

[−] **fn new() -> Counter**

Creates a new counter initialized to 0

[−] **fn get_value(&self) -> u64**

Get the current counter value

[−] **fn incr(&mut self)**

Increments the counter value

NOKIA bell labs

The Rust language

Advanced features

Advanced features
A powerful syntax inspired by functional languages

- Closures

```
let array = [1, 2, 3, 4, 5];
let new_array : Vec<i32> = array.iter()
                               .map(|v| v*2) // This is a closure
                               .collect();
println!("New array = {:?}", new_array);
```

- Type inference

```
let new_array : Vec<_> = array.iter().map(|v| v*2).collect();

let new_array = array.iter().map(|v| v*2).collect::<Vec<_>>();
```

```
// Both are valid
// x and y are both i32
let x: i32 = 12;
let y = 12;
```

- No NULL pointer, only Optional types

Enum std::option::Option

```
pub enum Option<T> {
    None,
    Some(T),
}
```

```
pub fn process(counter: Option<&mut Counter>) {
      if let Some(cnt) = counter {
            cnt.incr();
      }
}
```

```
process(None);
process(Some(&mut counter));
```

Advanced features
A powerful syntax inspired by functional languages

- Pattern matching and destructuring

```
pub enum Stuff {
	Count(Counter),
	Str(String),
	Nothing
}

pub fn get_some_stuff(stuff: Stuff) -> String {
	match stuff {
		Stuff::Nothing => format!("Got nothing"),
		Stuff::Str(s) => format!("Got a string: {}", s),
		Stuff::Count(cnt) => format!("Got a count: {}", cnt.get_value())
	}
}
```

```
let i = 12;
match i {
	0...5 => println!("Small"),
	e @ 5...15 => println!("Not so small: {}", e),
	e if e < 0 => println!("Negative"),
	_ => println!("Big")
}
```

```
let b1 = true;
let b2 = false;
let r = match (b1, b2) {
	(true, false) => "foo",
	(false, true) => "bar",
	_ => "baz"
};
```

Advanced features

A powerful syntax inspired by functional languages

- Trait-based genericity

```
pub trait Talk {
      fn say_hello(&self);
      fn say_goodbye(&self);
}
```

```
pub struct French;

impl Talk for French {
      fn say_hello(&self) {
            println!("Bonjour");
      }

      fn say_goodbye(&self) {
            println!("Au revoir");
      }
}
```

```
pub struct English;

impl Talk for English {
      fn say_hello(&self) {
            println!("Hello");
      }

      fn say_goodbye(&self) {
            println!("Goodbye");
      }
}
```

```
// Use trait object: a vtable is involved
pub fn talk(talker: &Talk) {
      talker.say_hello();
      talker.say_goodbye();
}

// The function will be implemented for each types
// it is called with
pub fn talk_no_cost<T: Talk>(talker: &T) {
      talker.say_hello();
      talker.say_goodbye();
}
```

```
let en = English;
let fr = French;
talk(&en);
talk_no_cost(&fr);
```

Advanced features
Error handling

- No exceptions, error handling based on returned values
- Standard library defines **Result<T, E>** which is returned by functions which may fail

```
pub enum Result<T, E> {
	Ok(T), // Contains the success value
	Err(E), // Contains the error value
}
```

```
pub fn may_fail(fail: bool) -> Result<i32, String> {
	if fail {
		Err("It failed".to_string())
	} else {
		Ok(12)
	}
}
```

- The result must be checked by caller

```
let mut res: i32 = may_fail(true).expect("Function failed"); // Will panic on error
res = may_fail(false).unwrap_or(13); // Unwrap the result, or use 13 on error
// Plenty of other possibilities
```

- The **try!** Macro or the **?** operator can be used to propagate the error

```
pub fn may_fail2(fail: bool) -> Result<i32, String> {
	let res: i32 = may_fail(fail)?;
	println!("Function did not fail");
	Ok(res)
}
```

```
pub fn may_fail2(fail: bool) -> Result<i32, String> {
	let res: i32 = try!(may_fail(fail));
	println!("Function did not fail");
	Ok(res)
}
```

Advanced features
Powerful macros

- Macros let us extend syntax by manipulating the AST
- For example we could simplify commonly written code like

```
use std::collections::HashMap;
let mut map = HashMap::new();
map.insert("foo", 1);
map.insert("bar", 2);
```

Into

```
let map2 = map! {
      "foo" => 1,
      "bar" => 2,
};
```

by implementing a **map!** macro

```
macro_rules! map {
      ($($key:expr => $value:expr,)*) => (
            {
                  let mut map = std::collections::HashMap::new();
                  $(map.insert($key, $value);)*
                  map
            }
      )
}
```

Advanced features

FFI: Simple and efficient C bindings

```rust
extern crate libc;
use libc::size_t;

#[repr(C)]
pub struct MyData {
	id: u8
}

// Redefine signature for functions from
// mylib.dll, mylib.so, libmylib.dll or libmylib.so
#[link(name = "mylib")]
extern {
	// Extern functions are unsafe by definition
	// and would require a safe wrapper
	fn my_c_function(data: *mut MyData) -> size_t;
}
```

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct my_data {
    unsigned char id
} my_data;

size_t my_c_function(*my_data data);
```

- **Bindgen** is a tool which generates Rust bindings from C headers

NOKIA bcom

The Rust Language

The borrow checker

The borrow checker
What is it

- One of the most important and powerful feature
- Check your code for a set of rules
- Checks are performed at compile time and has no runtime cost
- Prevents bad usage of memory
  - Use after free
  - Use after move
- Introduce notions of ownership, borrowing and lifetime
- You will love it, and you will hate it
- But hopefully, compiler errors are really friendly

The borrow checker
Ownership

- Variable bindings have **ownership** of what they're bound to
  - Moving the value will transfer ownership, preventing the **use after move**

```
let a = vec![1, 2, 3];
let b = a;
a.get(0)
```

```
error[E0382]: use of moved value: `a`
   --> src\main.rs:198:5
    |
197 |     let b = a;
    |         - value moved here
198 |     a.get(0);
    |     ^ value used here after move
    |
    = note: move occurs because `a` has type `std::vec::Vec<i32>`, which does not implement the `Copy` trait
```

  - Valid code would be
    - Either use b instead of a

```
let a = vec![1, 2, 3];
let b = a;
b.get(0);
```

    - Or clone a into b

```
let a = vec![1, 2, 3];
let b = a.clone();
a.get(0);
b.get(0);
```

The borrow checker
Borrowing

- With ownership comes the borrowing
  - Getting a reference (a pointer) to an existing binding
  - A reference cannot outlive the value it points to
  - A variable can have many immutable borrowers but only one mutable borrower
  - Cannot move a value while it is borrowed

```
let a = vec![1, 2, 3];
// b is a reference to a
let b = &a;
a.get(0);
b.get(0);
```

```
let mut a = vec![1, 2, 3];
let b = &mut a;
a.push(0);
b.push(4);
```

```
let mut a = vec![1, 2, 3];
let b = &mut a;
a.get(0);
b.push(4);
```

```
error[E0499]: cannot borrow `a` as mutable more than once at a time
   --> src\main.rs:206:5
    |
205 |     let b = &mut a;
    |                 - first mutable borrow occurs here
206 |     a.push(0);
    |     ^ second mutable borrow occurs here
207 |     b.push(4);
208 | }
    | - first borrow ends here
```

```
error[E0502]: cannot borrow `a` as immutable because it is also borrowed as mutable
   --> src\main.rs:205:5
    |
204 |     let b = &mut a;
    |                 - mutable borrow occurs here
205 |     a.get(0);
    |     ^ immutable borrow occurs here
206 |     b.push(4);
207 | }
    | - mutable borrow ends here
```

The borrow checker
Lifetimes

- Each reference has an attach lifetime
- Most of the time it's implicit, but sometimes it can (must) be explicit

```
let b = {
        let a = vec![1, 2, 3];
        &a
};
```

```
error: `a` does not live long enough
   --> src\main.rs:213:5
    |
212 |         &a
    |          - borrow occurs here
213 |     };
    |     ^ `a` dropped here while still borrowed
214 | }
    | - borrowed value needs to live until here
```

```
struct MyStruct {
        s: String
}
/// In that case, 'a is the lifetime. It's optional,
/// and given only for example.
/// In this example, the function returns a
/// reference to a string borrowed from the argument.
/// The argument will stay borrowed until
/// the string reference is dropped
fn get_string<'a>(val: &'a MyStruct) -> &'a String {
        &val.s
}
```

```
/// In that case, the struct contains a borrowed
/// data and has a lifetime constraint meaning
/// it cannot outlive the borrowed data
struct Example<'a> {
        ptr: &'a String
}
```

NOKIA bcom

The Rust language

Memory management

Memory management
RAII and destructors

- No garbage collector → No GC pauses
- Memory management is based on scopes
  - When a variable reaches end of scope its memory is freed
  - If the variable still owns its value (the value has not been moved), and if it has a destructor, the destructor is called
  - A destructor is defined by deriving from the trait **Drop**

```
struct Example;

impl Drop for Example {
	fn drop(&mut self) {
		println!("Destructor called");
	}
}
```

Memory management
The heap vs the stack

- Data can be stored on the stack

```
pub fn fn_ptr(ptr: &Counter) {}

let cnt_stack = Counter::new();
fn_ptr(&cnt_stack);
```

- Or on the heap through the Box type

```
let cnt_heap = Box::new(Counter::new());
fn_ptr(&cnt_heap);
```

  - Box::new() allocates memory on the heap
  - While the destructor (drop()) frees it

- **Box<E>** can be dereferenced to E thanks to **Deref** trait

```
let cnt_moved : Counter = *cnt_heap;
```

This moves the value out of the heap to the stack, freeing the allocated heap space

- Ref Counted (**Rc**) and Atomically Ref Counted (**Arc**) references are available is std lib

Memory management
Unsafe code

- Sometimes the compiler may not understand all the logic
  - Pointer arithmetic
  - Low level memory manipulation
  - Foreign Function interfaces
- Developers need a way to tell it « Hey, trust me on this » and deactivate some constraints
- The **unsafe** keyword let us mark a block of code or a function as doing some unsafe things
  - An unsafe block can cause segfaults if not properly written
  - Some functions are unsafe, and cannot be called outside of an unsafe block
  - Extern functions are unsafe by definition
- Raw pointers exists and have the exact same memory representation than references
  - ***mut T** is the equivalent of **&mut T**
  - ***const T** is the equivalent of **&T**
  - Converting from reference to raw pointer is safe, the other way is not

Conclusion

- Pros
  - An expressive language with little overhead
  - Fit for real-time applications development
  - Fit for embedded software development
  - Memory safety makes it a good candidate for critical applications
  - While it’s a bit hard to learn, it’s also a lot of fun
  - The Rust community is very welcoming and tolerant
- Cons
  - The learning curve is quite steep
  - A majority of third-party libraries are still immature
  - A permanent fight against the compiler

pierre-henri.symoneaux@nokia.com